AQA A Level Computer Science NEA

Luka Warren

2023

# Analysis

## Background

There exists online many popular "remixes" where someone has taken an existing song and distorted it, usually by adjusting its speed, bass and pitch, to achieve a desired effect. Often these versions of the song are preferred to the originals when used to accompany short-form content on video-sharing sites such as TikTok. For example, the song "Money Trees" by Kendrick Lamar has 120 million views on YouTube in its original form, but also a sizeable 1.6 million views on one "TikTok remix" alone, and indeed on TikTok itself it is rare to hear the original version.

As such it can be seen that there exists a large audience of people who enjoy listening to altered versions of popular songs. However, the most popular music listening programs, including Spotify and YouTube music, provide no mechanism for manually altering songs by one’s self. Whilst there exists ready-made "remixes" by others, there are no mainstream programs which allow a user to "remix" a song in real-time as it is being listened to. This presents two main problems:

* Many songs do not have any accompanying "remixes" to satisfy a user’s need
* The barrier to entry for creating a "remix" prevents easy and user-friendly experimentation

It is therefore the aim of this coursework to create a system to allow users to "remix" songs in real-time as they listen to them. Such a system would allow for comprehensible, user-friendly experimentation, removing the above barrier to entry, whilst providing a mechanism to alter the sound of any arbitrary audio, removing the need to rely on other’s work.

However, a system can only be comprehensible and user-friendly if the exact needs of the users themselves are known, and so first a representative sub-section of the user-base must be interviewed.

## Collection of Data

A number of interviews were conducted with peers that self-reported to enjoy listening to song "remixes". Below is a brief summary of the questions asked and the relevant responses.

#### Question 1 - Why do you sometimes prefer a song’s remix?

##### Student 1

Videos on TikTok are only about 30 seconds long. It’s good to speed up a song because otherwise you couldn’t enjoy the full chorus.

##### Student 2

I’m not sure really - I think I prefer it when a song has more bass than usual, and I like how distorted it sounds.

##### Student 3

I like watching remixes on YouTube because when they visualise the music it always looks very cool.

#### Question 2 - How does a remix typically differ from the original song?

##### Student 1

Usually they’re faster and I guess that makes them higher-pitched too.

##### Student 2

They have more bass and are slowed down a bit. They have a bit of an echo [effect] too, and sometimes also they add noise to make it more relaxing. Even the volume is different.

##### Student 3

A remix usually has a different speed and is higher or lower than the original.

#### Question 3 - What features would you like in a real-time audio editing program to assist in "remixing" music?

##### Student 1

I’m not really good with editing so hopefully it would be easy to use. I’d probably only want to apply the same effect every time so there should be a way to help me with that.

##### Student 2

I’d like to be able to apply it to my entire music collection because that way all my songs could have the same effects applied. In other words I’d like to apply it to my music playlist.

##### Student 3

I think it would be cool to change the speed of songs and change the bass and treble. My car stereo can do that and it’s really interesting to play with.

## Interview Interpretation

### Frequencies

Through the interviews it was understood that the ability to modify certain frequency ranges was desirable, in order to affect both the bass and treble. Typically, there are two main ways of doing this:

* Applying a low-pass or high-pass filter to broadly modify the frequencies represented
* Modifying the incoming audio in its frequency domain using a Fourier Transform, then converting it back to the time domain using an Inverse Fourier Transform

Whilst applying a low or high pass filter is a very inexpensive operation, it does not provide exact control over the frequencies modified. Hence in order to best be able to manipulate frequencies, a Fourier Transform must be used. This is not a computationally trivial task and so every effort must be made to ensure that the program created can still run fast enough to be real-time on typical high school hardware, so as not to increase the barrier of entry, as this would go against the intention of the project.

### Playback Speed

The system must also be able to alter the playback speed, as this feature was highly requested. However, again this must not conflict with the real-time requirements of the system on modest high school hardware. In other words, the adjusting of playback speed should not demand a significant computational overhead (or ideally any overhead at all).

### Other Audio Effects

Student 2 mentioned "remixes" often contain an "echo" or additional "noise", and so in order to avoid limiting the program to merely changing a song’s speed or frequency response, it should also allow the user to apply various audio effects, including but not limited to the ones above. In order to maximise the ease of experimentation, as is desired, the effects should be easily configurable.

### Visualisation

As the program is already meant to perform a Fourier Transform on the incoming audio data, it will thus already have a representation of the audio in the frequency domain. Hence it would be trivial to display this data graphically so as to provide an audio visualisation feature that is, computationally, essentially free. It is hoped that by providing visual feedback to the audio as it is edited, the effects of, for example, adjusting the bass, will be easy to see and thus the processing the software is carrying out will be easy to comprehend. The exact nature of this visualisation (that is: how audio is to be graphically displayed in frequency space) is to be covered in detail below in section 2 ("Design"). However, put simply, one can imagine a "bar chart", with frequency ranges on the x-axis, and amplitude on the y-axis. Such a visualisation would allow the user to easily see, for example, the immediate effects a "bass boost" would have on the frequency space of the audio being played.

## Fourier Technical Analysis

As mentioned above, in section 1.3.1, it is essential to use a Fourier Transform to be able to modify the incoming audio in frequency-space (for example to provide a "bass-boost"). However, the primary concern is if this algorithm can be carried out fast enough to be able to process audio in real-time.

The primary method for computing Fourier Transforms efficiently is using a Fast Fourier Transform (FFT), which is itself a subset of the Discrete Fourier Transform (DFT). DFTs operate on discrete packets of data, such as audio samples, as opposed to continuous waves, and are hence ideal for this project. Because of the various mathematical tricks used in FFTs, they reduce the time-complexity of computing DFTs from to , providing the performance this project needs (as when dealing with large audio samples N will typically be large).

The most popular FFT algorithm is the "Cooley-Tukey" algorithm, which recursively breaks down its input into two halves. The only limitation of this approach is that the input data size must generally be a power of two, but as the program should have control over how much data it processes at a time, this should not be an issue.

The formal definition of a DDT may look daunting:

However, a Fourier Transform is essentially just a process to convert a signal into its constituent sine waves. For example, a very basic song may be composed of a number of sine waves with low frequencies (i.e. the bass) plus many higher frequency waves that combine to form the vocals and instruments. Each frequency in an FFT has a corresponding amplitude (volume) and phase (position in time). By adjusting the amplitudes, the relative volumes of a song’s frequencies can be modified at will.

It therefore appears that the issue of manipulating frequencies in the coursework should be technically possible, providing performance is maximised by using a Cooley-Turkey FFT.

## Programming Language and Performance Technical Analysis

Before embarking on a project design, an appropriate programming language must be chosen. By considering the project requirements it is apparent that two main groups of languages will likely prove insufficient.

#### Interpreted Languages

Languages such as Python and Ruby, whilst intuitive and easy to use, may not provide sufficient CPU performance to easily allow the system to be real-time. Most music typically has two channels, each at a sample rate of around 44,000 Hz, meaning the system must process roughly 88,000 floating point numbers per second at minimum. This number can quickly grow if, for example, an echo is required, as then multiple seconds of audio may need to be considered. Whilst most machines have CPUs powerful enough to accomplish this even when under an interpreted language, it may result in high CPU usage and significant energy requirements, raising the barrier of entry to using the program. Ideally, the program should therefore not use an interpreted language, so as to maximise the number of people who can run it.

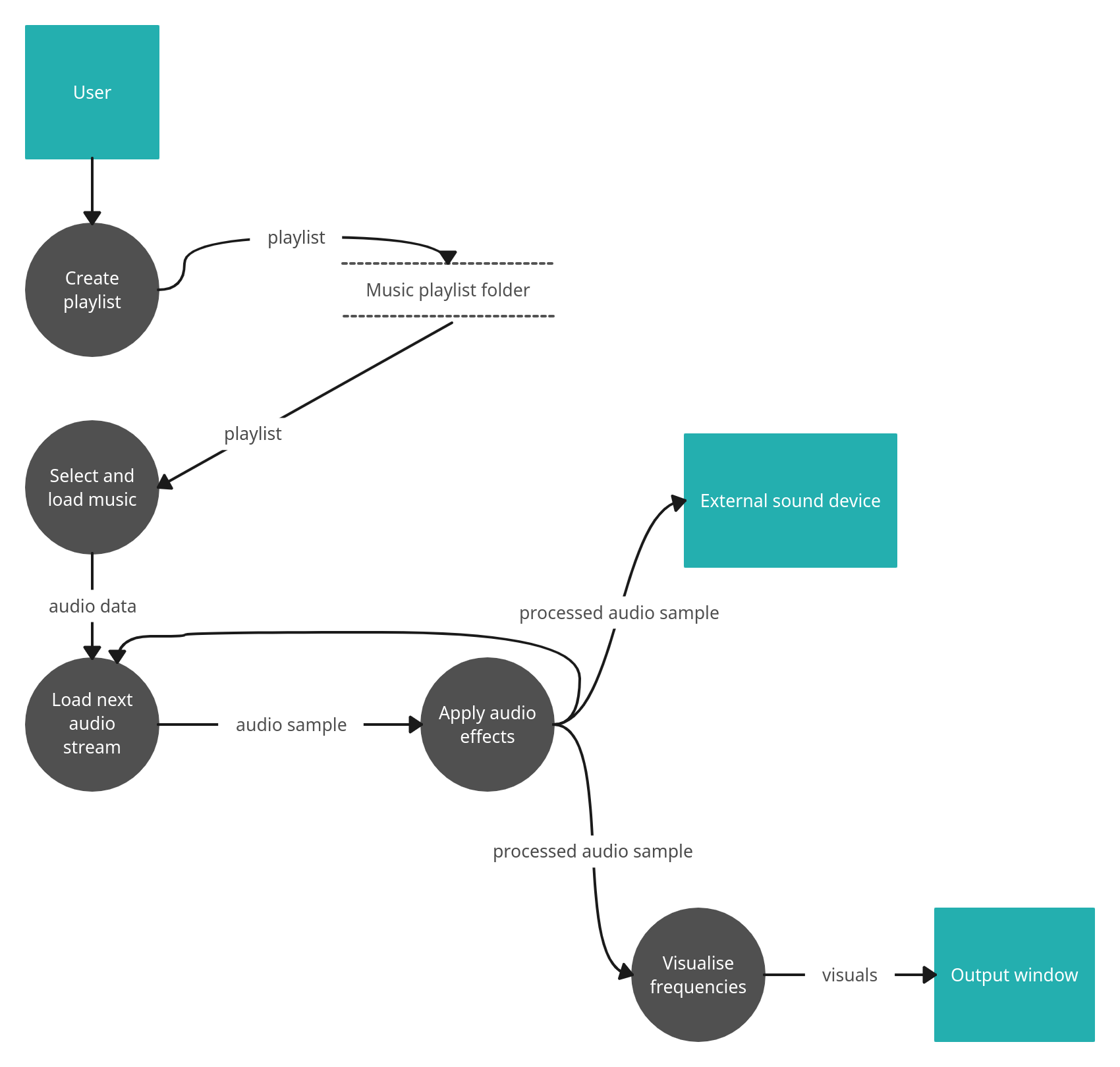
#### Garbage-collected and JIT languages

Languages such as C# and Java are both Just-In-Time (JIT) compiled and garbage collected. This is unacceptable for a real-time audio processing application as both JIT compilation and garbage collection typically introduce frequent micro-stutters, which may result in occasional blips in the program’s audio, ruining the output.

After excluding the two groups of languages above, it can be seen that any language chosen must be compiled ahead-of-time to native machine code to maximise performance and avoid JIT stutter, whilst also providing direct control over memory to avoid the garbage-collection issues described above. Ideally, it should also be a modern language capable of OOP. After reviewing this requirements I have decided to use C++, as I am extremely familiar with the language and believe it suits all these requirements.

## Data Flow Diagram

After considering the requirements of the project, I believe the following should serve as a good model of how data should be treated.



The data flow diagram (DFD)

## Analysis of Similar Software Products

Every consideration must be given to the existence of other software products which may implement, either partially or fully, the features of the program yet to be implemented.

There exists two broad categories of software which provides features similar to this project:

* Real-time music players - the majority of music listeners listen to music in real-time using streaming services such as Spotify or Apple Music, which can provide certain features to adjust songs in frequency-space (albeit crudely)
* Audio editors - for those who wish to radically alter audio (e.g. to create a remix), software such as Audacity provides many effects and processing opportunities. These are, however, "offline" in the sense that such software is not real-time.

### Streaming Services

Currently within the market the two most popular music streaming products are Apple Music, with 88 million customers in 2022, and Spotify, which over 500 million. These allow users to listen to music on-demand, with both providing an "equaliser" feature that allows the user to modify the relative volume of frequencies within a song in real-time. In other words, with both these products one can, for example, perform a "bass boost", which mirrors very closely one of the main aims of this project.

|  |
| --- |
|  |

The Spotify equaliser

|  |
| --- |
|  |

The Apple equaliser

As can be seen above, there are a number of similarities with the aims of this project:

* The user can easily drag various points to modify the relative volumes of a number of "frequency groups". For example, one could drag the left-most dot to its maximum height in both to boost frequencies under either 60 or 30 Hz.
* Ready-made "presets" can be chosen at will to assist those who may be unfamiliar with the settings presented or want to achieve a particular, pre-made effect.
* In Apple Music, the overall volume of the music can be adjusted.

However, there are also a few notable areas in which the two products above fall short of the features being implemented in this project, and hence fail to satisfy the needs of the users outlined in section 1.1.

* In neither case does the user have exact control over the precise frequencies being modified. If, for example, one had identified a particularly intriguing instrument playing from 500 Hz - 700 Hz, it would be impossible to isolate that frequency and boost it.
* There is no option to apply other audio effects such as an echo or noise. As such there are very few ways one can actually modify the audio, so creating a full "remix", that feels distinct from the original, is impossible.

Thus whilst on a surface-level, the presence of "audio equalisers" in both software products may appear to conflict with the aims of this project, especially as they perform their processing in real-time, closer inspection reveals that they offer only extremely limited options, without the ability to customise which precise frequencies are adjusted or indeed apply any other audio effects.

### Audio Editing Software

On the other hand, there exists many programs which allow a user to modify audio using a great range of effects and filters. As the aim of this project is to lower the barrier of entry to creating song "remixes", it is to be free, and as such it is most worthwhile only considering other free audio editing software.

The most popular free audio editing application is Audacity, an open-source project that supports an extremely large number of effects. Additionally, it supports viewing a spectrogram of the audio being played (which visualises the audio in frequency-space just like this project aims to do), although the feature is somewhat hidden away.

|  |
| --- |
|  |

Audacity when configured to display a spectrogram of the audio, seen as two multi-coloured strips (one for each channel)

|  |
| --- |
|  |

Audacity’s "effects menu", offering a number of categories which themselves all contain numerous effects

|  |
| --- |
|  |

Audacity offering a menu to configure the application of a computationally-intensive, though highly accurate, reverb effect

As can be seen above, Audacity thus presents a number of strengths which could be seen to risk overshadowing this project:

* A great number of effects are supported, far greater than this project’s scope could hope to provide
* The effects are all highly physically accurate, with extreme levels of customisation

However, its weaknesses should also be stressed:

* The user interface is daunting for new users, raising the barrier of entry. For example, the process of displaying a spectrogram is not at all obvious and is hidden away.
* The sheer number of parameters for controlling effects is likely very daunting for people that just want a "quick remix".
* There are no ready-made "presets" for those too confused to immediately start applying effects themselves, or for those quickly looking for a specific modification.
* One cannot apply any effects in real-time, and often the effects are so computationally expensive that one must wait multiple seconds before hearing the result, even on powerful hardware.
* It is impossible to create a "music playlist" to listen to multiple songs in quick succession.

Hence it can be seen that whilst Audacity, and other audio editing software, provides many mechanisms for manipulating audio, it cannot be done in real-time, and often powerful hardware is required. They are also completely unsuitable as real-time music players, as they cannot play "playlists". Additionally, the sheer range of editing options would likely prove daunting to even the most experienced user, and so when one considers the high barrier of entry, the failure to be real-time, and the lack of "playlist" functionality, it is clear that audio editing programs such as Audacity do not conflict with the aims of this project.

### Summary of Similar Software Products

|  |
| --- |
|  |

S.W.O.T diagram for streaming services

|  |
| --- |
|  |

S.W.O.T diagram for audio editing prograrms

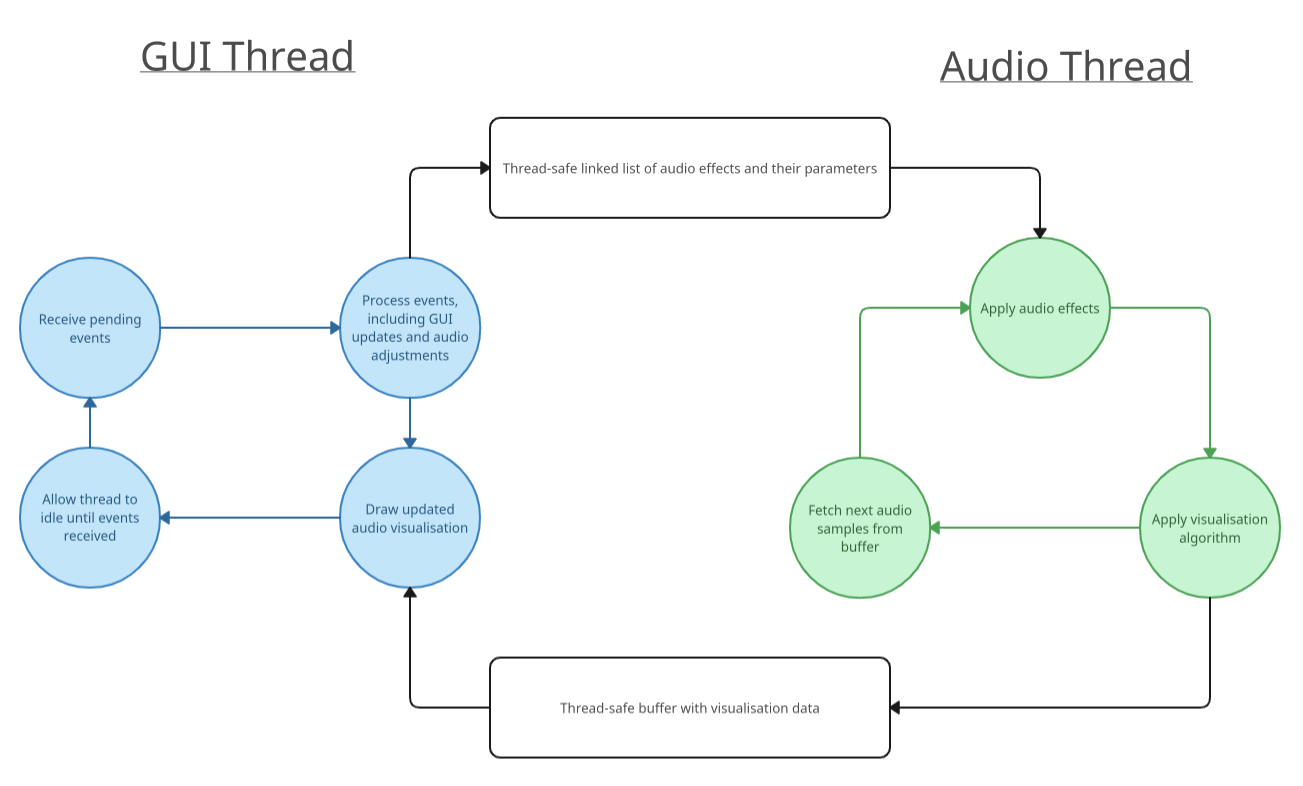
## Final objectives

1. The user must be able to load a collection of audio files known as a "playlist" and then play the audio files contained within in a logical order
2. The user must be able to visualise the current audio being played in the frequency domain (i.e. visualise the frequencies)
3. The user must be able to modify the audio’s frequency domain (i.e. selectively modify frequencies such as by performing a bass boost)
4. The user must be able to apply additional "audio effects" to further enhance the music: echo, volume adjustment and noise
5. The user must be able to configure these "audio effects" individually and also apply pre-made "presets" to quickly reach a desired effect
6. The system must run in real-time on an average school computer

# Design

## Multithreading

In order to maximise ease-of use, the software should have a graphical user environment (GUI) so that the current audio being played can be easily visualised (in-line with objective 2). The program will use a multithreaded model, with a separate "audio thread" and "GUI thread", allowing the two to run concurrently without blocking each other’s processing.



Inter-thread diagram - see below for justification for thread-safe data structures

Typically, GUI programs are written using an event-based paradigm that minimises CPU idle-time. The consequence of this is that, for most of the time, the GUI thread is suspended, awoken only when events from the user (such as mouse clicks or resizing the window) "wake it up". This is desirable in order to minimise system resources used, as more CPU-time will be available for the audio processing requirements, helping to reach the real-time requirements of objective 6. However, this presents a unique challenge. With a single-threaded model:

* If the event-based model is followed, the GUI thread is only active when there are pending GUI events to be processed, meaning audio processing can only occur sporadically (resulting in "non-constant" audio)
* If instead the GUI thread is constantly active processing audio it will never reach a point where it can process pending events, meaning the program will hang and refuse to process inputs.

Hence it is desirable to split the program into two distinct threads. The audio thread can play the audio and perform all necessary processing tasks, whilst the GUI thread can relay input parameters and commands to the audio thread (such as "switch song", "apply effect", etc.).

To avoid race conditions[[1]](#footnote-91), the data that is read by both threads should be thread-safe - only one thread should be able to access the data at a time. This can be achieved by using mutexes[[2]](#footnote-92).

### Audio Effects Data Structure

#### Picking a data structure

The user will likely want to adjust the order of audio effects at will, and as the same time, it must be very fast to insert and remove audio effects so as to minimise the time spent not processing audio (even a very short pause may result in "crackles" on weaker hardware). To solve this problem, the audio effects can be stored in a linked list, as unlike std::vectors (dynamic C++ arrays) they prove fast insertion, deletion and swapping irrespective of the number of elements stored.

#### Making it thread-safe

To satisfy the requirements of multithreading (see above), I will write my own custom "atomic linked list", backed by a mutex[[3]](#footnote-95), which will function just like a normal linked list but maintain thread-safety in all its operations.

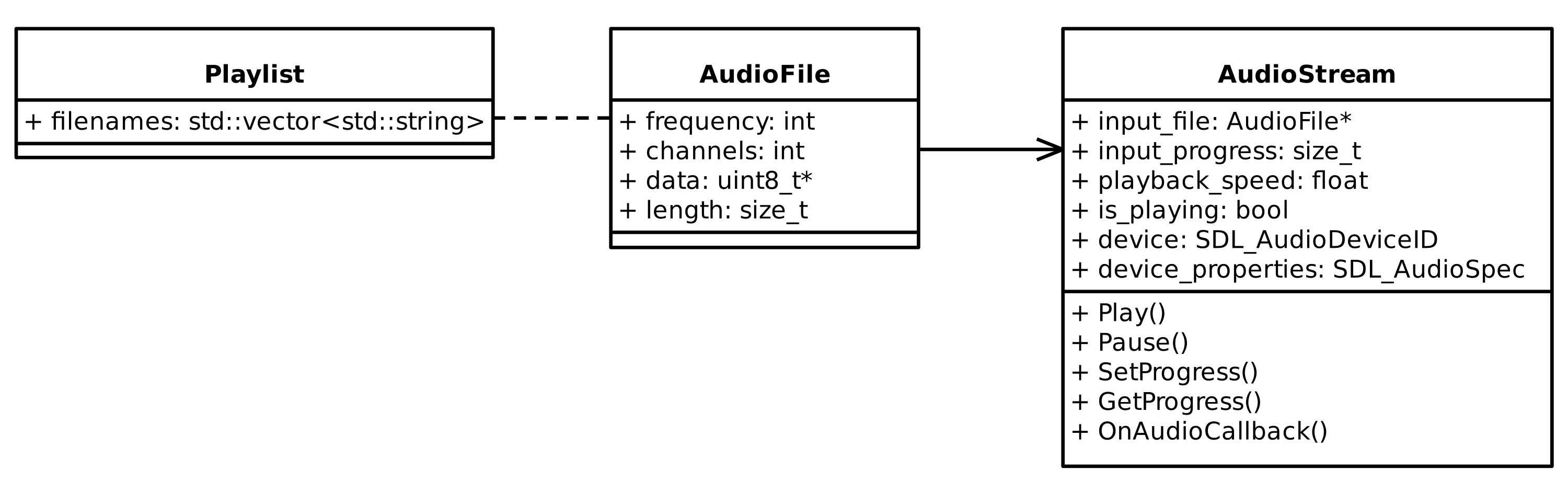
## Audio Data and Playback

### Audio Data

The program will have to load a variety of user-supplied data in order to operate:

* As described in objective 1, the user must be able to load a collection of audio files known as a "playlist", which will contain the paths of one or more audio files on the system.
* Each audio file will consist of a number of audio samples, which will need to be loaded into memory when needed, then freed when not in use.
* Audio files also contain other crucial information, such as the audio frequency (e.g. 44,000 Hz), the number of channels (usually mono (1) or stereo (2)), and the number of samples (the "length").
* Thus to keep track of the audio files loaded into memory, each audio file will need to store its raw audio samples, its frequency, the number of channels, and the number of samples.

### Audio Data UML



UML for Audio Data and IO - separate getter and setter exists for AudioStream progress as the caller will express progress as percentage (e.g. 50% played) so that it is independent of file size. The playlist contains the filenames of all audio files on disk, and an AudioFile instance is created, when needed, by loading the audio file from disk using this filename.

### The Need for Streaming

It may be tempting to load and unload all audio data in a hierarchal fashion as such:

1. An attempt is made in the code to load a playlist.
2. To do this, the playlist will be read from disk and all the audio file paths contained within will be loaded into memory.
3. Each audio file path will be verified to check it is valid exists on the system.
4. If the playlist is valid, each audio file will then be loaded using the paths provided.
5. Thus the loading of a playlist will involve loading all audio files referenced within.
6. At the end of the program, when the playlist is no longer needed, all the playlist’s audio files will be freed from memory, followed by the playlist itself.

However, this is not a practical approach due to memory usage constraints. If a user attempted to load a playlist consisting of 200 songs, each 5 MB each, this would consume roughly 1000 MB of memory for the entire duration of the program, even though only 1 audio file can be played at once (and hence only one needs to be in memory at any one time). This conflicts with objective 6 ("the system must run in real-time on an average school computer") as many computers may not have large amounts of free memory, particularly if other programs are running, which may lead to an out-of-memory crash.

Instead, I have decided to "stream" audio files as they are played, so that only the audio file currently needed is resident in memory. This can be modelled as followed:

1. An attempt is made in the code to load a playlist.
2. To do this, the playlist will be read from disk and all the audio file paths contained within will be loaded into memory.
3. Each audio file path will be verified to check it is valid exists on the system. If the playlist is valid, the execution of the program will continue.
4. Each time the next audio file is to be played from the playlist, the program will dynamically load it from disk (using the path from the playlist) and store it in memory.
5. When the next audio file is chosen, it is loaded as described above. Crucially however, the previous audio file is first unloaded from memory, as it is no longer needed.
6. At the end of the program, the currently playing audio file and playlist are both freed.

In this way, the issue of large playlists resulting in extremely large memory consumption will be avoided, as only one audio file will be loaded at once.

### Audio Playback

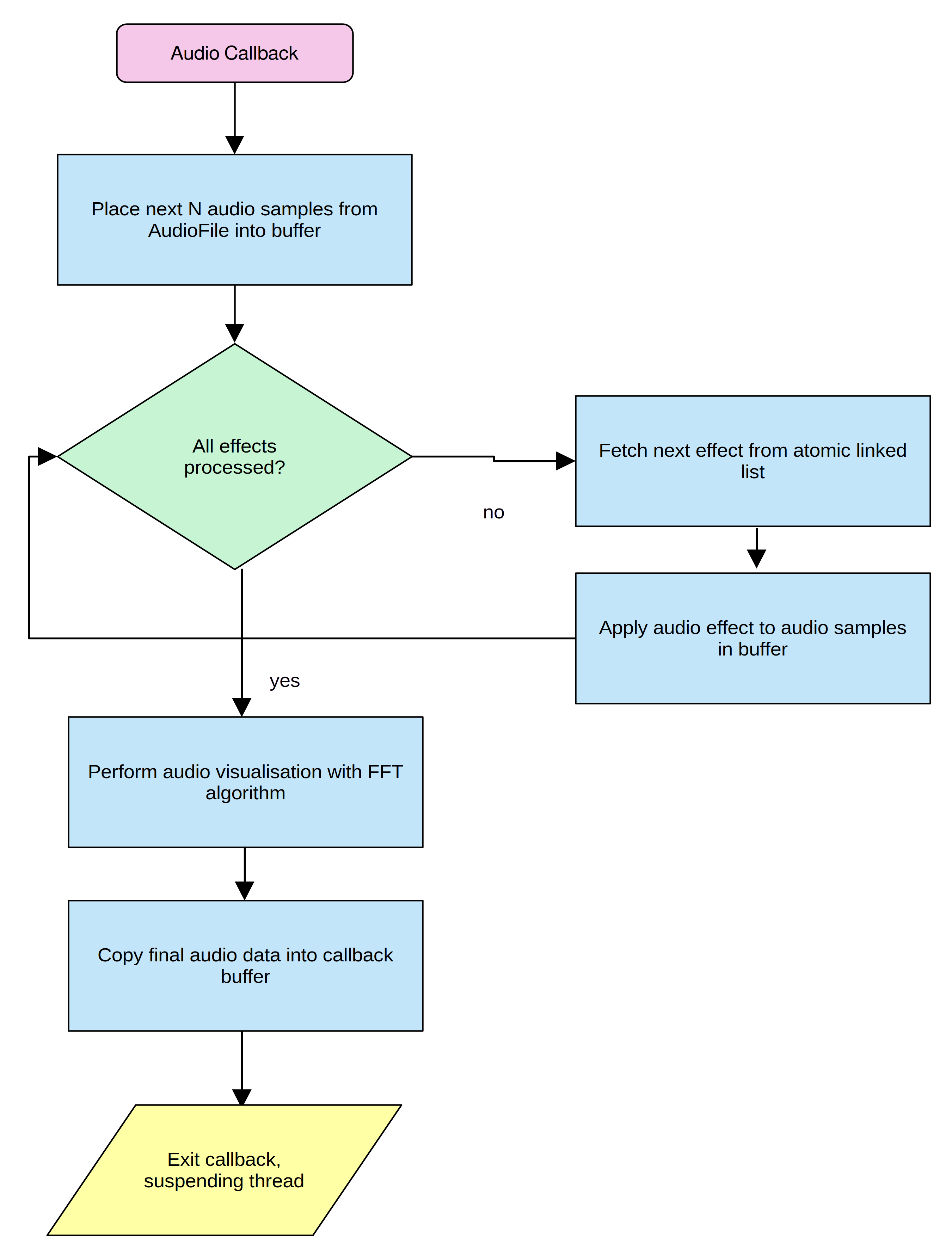
The playback of audio itself presents many challenges. In order to make the code as modular and decoupled as possible, I will abstract away the low-level creation of audio devices, pausing, un-pausing, etc. into an "AudioStream" class. One will simply create an "AudioStream", supply it with data, and the class will manage the various complexities of multithreading and feed the audio buffer with data at the appropriate times.

In order to maximise portability of the code, and hence make it as cross-platform as possible in order to maximise the program’s audience, I have decided to use a library called "SDL2" to handle audio playback, as it abstracts away the native APIs one would have to otherwise use. In this way, separate audio code does not have to be written for Windows, Linux, etc.

The code that plays audio on the system will run on a separate thread (see multithreading section). This audio thread is invoked at regular intervals by the operating system by way of a "callback" function. When this happens, it is the program’s responsibility to supply the operating system with the next buffer of audio. This is summarised below:

1. An "AudioStream" is created and supplied with the raw audio samples from the audio file, as well as pointers to the atomic linked list of audio effects and to the visualisation data buffer.
2. The AudioStream uses SDL2 to invoke audio playback at regular intervals on a separate thread (the "audio thread") using a callback
3. Every time the callback is called, the AudioStream will fetch the next section of upcoming audio that it has been supplied with.
4. Each audio effect will then be applied (using the audio effects atomic linked list).
5. The audio visualisation module will then be invoked on the audio just processed, and its output written to the visualisation data buffer.
6. Now that all work is done for the current section of audio, the processed audio will be copied to the callback’s audio buffer and the audio thread will suspend itself.
7. When the next section of audio is due, the callback will be re-invoked.

### Audio Playback Flowchart



Flowchart for AudioStream’s SDL audio device callback

## Audio Effects Architecture

The full list of audio effects detailed in the analysis section is as follows:

* Equaliser (frequency modification) - selectively modifying frequencies such as by performing a bass boost
* Echo - making audio sound like it’s recorded in a large room
* Volume adjustment - modifying the amplitude of the audio
* Noise - adding subtle imperfections to the audio

### Unique Audio Effect Traits

Each audio effect will need its own properties, and potentially its own mutable state (for example, the echo effect needs to "remember" the previous audio samples so it can repeat them later). Below is a summary of the requirements, properties and state of each audio effect.

| Effects | Requirements | Properties | State |
| --- | --- | --- | --- |
| Equaliser | Allow the user to alter the volume of a selected frequency range. Multiple equaliser effects can be applied successively to cover multiple ranges. | Lower frequency Upper frequency Multiplier | None |
| Echo | Produce an echo effect where audio sounds like it’s getting reflected in a large room | Fall-off (how quickly echoes fade) Delay samples (how many samples must pass before a sample is echoed) | Previous audio samples buffer |
| Volume | Adjust the volume / amplitude of incoming audio | Volume multiplier | None |
| Noise | Add subtle imperfections to the audio | Intensity (the volume of the noise) | None |

### Common Audio Effect Traits

Immediately it is obvious that all audio effects will share many common features. Each effect shall:

* Take a number of audio samples as input
* Have a number of configurable options which need to be exposed to the GUI front-end
* Perform processing on all audio samples at once
* Output its final processed audio

Given these requirements it is wise to use an object-orientated inheritance approach where effect subclass inherits from a common parent, which provides common functionality (such as the storage and exposing of configuration options), as well as providing a common interface that other parts of the code can use. In order to abstract away the details of interacting with an audio effect, two new classes will also be needed.

##### Packet

A packet represents a chunk of audio awaiting processing by the effect. However, some effects require knowledge of both the previous audio samples and future audio samples (like echo). Thus, each packet will consist of 3 audio buffers - one for the previous, current and next buffer of audio samples. A packet will also contain the frequency of the incoming audio as this is required for the FFT maths.

##### Property

Each audio effect has a number of configurable properties. To aid in validation, each property will have a current, minimum and maximum value, as well as a flag to differentiate between integer values and decimal values.

Both these classes will be independent of the main audio effect parent class. The audio effect parent class will merely use these classes to represent the data provided and stored by it. This is an example of *composition*.

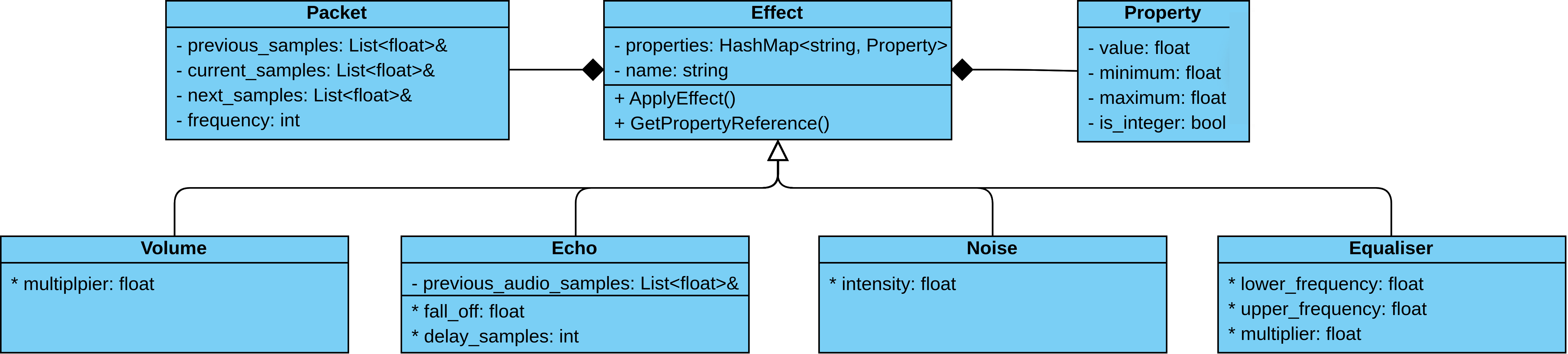
### Properties Data Structure

Each audio effect will have a list of properties. When drawing a properties GUI, the front-end code will want an easy and convenient way of both getting a list of all available properties, along with their respective names. In a similar fashion, when setting the values of certain properties, it will be most convenient if properties can be accessed using their names.

I will use a hash-map as my data structure for this purpose. Hash-maps can be indexed by using the name of the property (e.g. "minimum frequency" as the key), simplifying the front-end code and avoiding the need for a separate "property name" variable. In addition, they provide very fast look-up with an O(1) time. Whilst hash-maps are expensive when it comes to adding and removing elements, each audio effect will only have a fixed number of properties so this will not be an issue.

### UML Class Diagram

After considering all the requirements of the multiple classes required, I have constructed a class diagram. However, a slight modification has been made to the typical UML structure: properties of audio effects are prefixed with an \* (asterisk) in order to indicate that their are not attributes *per se*, but rather are elements in their parent’s properties hashmap.



UML class diagram containing attributes, operations and custom "properties"

## High-Level Audio Effects Flowcharts

|  |  |
| --- | --- |
| Equaliser |  |

Flowchart for equaliser (frequency modification) audio effect

|  |  |
| --- | --- |
| Echo |  |

Flowchart for echo audio effect

|  |  |
| --- | --- |
| Volume |  |

Flowchart for volume audio effect

|  |  |
| --- | --- |
| Noise |  |

Flowchart for noise audio effect

## Audio Visualisation and Fourier Algorithm

As can be seen in the equaliser flowchart, there is a need to perform both a Fourier Transform and an Inverse Fourier Transform on audio data in order to manipulate the audio in its frequency domain (i.e. to manipulate certain frequencies in isolation). In addition, as per the objectives outlined above, the program has a need to use this data to visualise the audio being played ("the user must be able to visualise the current audio being played in the frequency domain").

Hence there is a need for pseudo-code to be designed to perform the three algorithms involved:

1. A Fast Fourier Transform to convert incoming time-domain audio to its frequency-domain
2. An algorithm that graphically displays this frequency-domain audio for the purposes of visualisation
3. An Inverse Fast Fourier Transform to convert frequency-domain audio back to the time-domain

### Cooley-Tukey FFT and IFFT Algorithm

#### Summary of Fourier Analysis from Section 1

Careful consideration of the problem in the analysis section revealed that acceptable performance could be achieved by using a Fast Fourier Transform (FFT), which used various mathematical techniques to reduce time complexity from to , greatly reducing computational overhead. The most popular FFT algorithm, the "Cooley-Tukey" FFT, was chosen, due to its use of recursion to both efficiently and elegantly perform the computations required.

#### Comparing the FFT and IFFT

It should be stressed that there is very little difference between performing a "normal" Fast Fourier Transform using the Cooley-Tukey algorithm and an *inverse* Fast Fourier Transform. For this reason, I have decided to combine the two into a single function to eliminate as much code duplication as possible.

#### Pseudo-code implementation of algorithm

-- Converts an array of floats (i.e. the audio samples) into an array of complex numbers  
-- by using 0 for the imaginary parts. We must check the incoming data is a power of 2  
-- as later code that uses the results of this computation (i.e. the FFT) depends on this.  
function convert\_samples\_to\_complex\_form(samples: array of float) -> array of complex:  
 if size of samples is not a power of 2:  
 raise error "FFT data has invalid size"  
  
 complex\_samples := new array of complex with size equal to size of samples  
 for each sample in samples:  
 append (sample, 0.0) as a complex number to complex\_samples  
  
 return complex\_samples  
  
-- Performs both either an FFT or an IFFT recursively by diving the data into two  
-- until the trivial base case is reached  
function do\_fft(input: reference to array of complex, mode: Mode):  
 N := size of input  
 if N <= 1:  
 return  
  
 -- Split data by even and odd indices  
 even := new array of complex with size N/2  
 odd := new array of complex with size N/2  
 for i from 0 to N/2 - 1:  
 even[i] := input[2\*i]  
 odd[i] := input[2\*i + 1]  
  
 -- Perform FFT / IFFT recursively on even and odd halves  
 do\_fft(even, mode)  
 do\_fft(odd, mode)  
  
 for k from 0 to N/2 - 1:  
 -- Perform FFT calculation by manipulating audio sample in the complex plane  
 -- as per the formal defition (see analysis section)  
 sign := if mode is Normal then -1.0 else 1.0  
 t := polar(1.0, sign \* 2.0 \* PI \* k / N) \* odd[k]  
 input[k] := even[k] + t  
 input[k + N/2] := even[k] - t  
  
-- Example code for using FFT  
incoming\_audio = fetch\_upcoming\_audio\_data\_from\_buffer()  
complex\_audio = convert\_samples\_to\_complex\_form(incoming\_audio)  
do\_fft(complex\_audio, Mode::Normal)  
  
-- (calculations on frequency domain, including visualisation, performed here)  
  
-- Convert back to time domain (example code using IFFT)  
do\_fft(fft\_result, Mode::Inverse)  
final\_audio = discard\_imaginary\_parts\_of\_complex\_array(complex\_audio)  
submit\_audio\_to\_speakers(final\_audio)

### Visualisation Algorithm

#### Interpreting the results of an FFT

After an FFT has been performed, the resultant frequency-domain data needs to be visualised. The data returned from an FFT is an array of complex numbers where each index corresponds to a particular range of frequencies. The amplitude of these frequencies is just the magnitude of the complex number at that index. For example, index 12 might correspond to frequencies from 100 Hz to 130 Hz, and the magnitude of the complex number at fft\_array[12] would be the amplitude.

The range of frequencies each element in the FFT data represents is called the "frequency resolution", whereby:

Hence for any given index in the array, the minimum frequency it represents is given by:

The visualisation I have chosen is a bar chart, where the x-axis represents frequency and the y-axis represents amplitude. Each bar can be said to have:

* a minimum frequency
* a maximum frequency
* an amplitude

Hence an algorithm is needed to convert the incoming FFT frequency-domain data into the above format for later rendering. The minimum and maximum frequencies can be derived from the equations above.

#### Using a non-uniform scale for frequency

The human ear does not perceive frequencies in a linear fashion. In other words, if one adds 500 Hz to a sound-wave repeatedly, the "jump" in pitch will not always sound the same. This is because human-hearing follows a roughly logarithmic scale when it comes to detecting frequencies. This represents a problem as using a uniform, linear scale on the visualisation bar chart, whilst mathematically correct, will not sound plausible to the ear. Instead, the x-axis (frequency) for the bar chart must be adjusted so that, relative to the human ear, each bar represents roughly the same jump in *perceived* pitch. The most suitable scale for this is the "Bark scale", where "equal distances correspond with perceptually equal distances" as described above. A frequency can be converted into its Bark equivalent using a simple formula:

#### Using a non-uniform scale for amplitude

Just like with frequency, the human ears perceive amplitude logarithmically too. If a sound-wave has twice the frequency, it will not necessarily sound twice as loud. Thus the y-axis of the bar chart must be adjusted to reflect the *perceived* loudness. A good approximation is to take each amplitude and log it to base 10.

#### Ignoring inaudible sounds

The human ear cannot hear sounds below 20 Hz or above 20,000 Hz. These frequencies should therefore be ignored in the visualisation, and hence the algorithm should be able to reject frequencies outside a specified frequency,

function hertz\_to\_bark\_scale(hertz: float) -> float:  
 return 13.0 \* arctan(0.00076 \* hertz) + 3.5 \* arctan((hertz / 7500.0) \* (hertz / 7500.0))  
  
-- GroupSettings is a data structure consiting of the number of samples ("buckets") provided to the FFT,  
-- the frequency of the incoming audio, and the minimum and maximum acceptable frequencies.  
function convert\_fft\_to\_bar\_chart\_format(fft: array of complex, group\_settings: GroupingSettings) -> array of FrequencyRange:  
  
 -- Work out frequency resolution  
 n\_samples := size of fft  
 frequency\_resolution := group\_settings.frequency / n\_samples  
  
 -- Work out min and max frequencies in Bark scale and distance between "buckets"  
 minimum\_frequency\_bark := hertz\_to\_bark\_scale(group\_settings.minimum\_audible\_frequency)  
 maximum\_frequency\_bark := hertz\_to\_bark\_scale(group\_settings.maximum\_audible\_frequency)  
 bark\_distance := (maximum\_frequency\_bark - minimum\_frequency\_bark) / group\_settings.n\_buckets  
  
 buckets := new array of float with size equal to group\_settings.n\_buckets, initialized with zeros  
  
 -- The FFT is symmetrical (because audio data lies only on the real axis), so we actually only need  
 -- to visualise the first half of it  
 for i from 0 to (n\_samples/2 - 1):  
 frequency := i \* frequency\_resolution  
 if frequency >= group\_settings.minimum\_audible\_frequency  
 and frequency <= group\_settings.maximum\_audible\_frequency:  
  
 -- Use Bark scale conversion to work out location of "bar" in bar chart  
 bark\_frequency := HertzToBarkScale(frequency)  
 index := (bark\_frequency - minimum\_frequency\_bark) / bark\_distance  
  
 if index < size of buckets:  
 -- Calculate amplitude of frequency (i.e. the magnitude of the complex number)  
 -- Add this to the height of the bar at this point in the bar chart  
 buckets[index] += absolute value of fft[i]  
  
 -- Work out maximum amplitude in bar chart  
 max\_magnitude := 0.0  
 for each bucket in buckets:  
 if bucket > max\_magnitude:  
 max\_magnitude := bucket  
  
 -- Scale magnitudes logarithmically  
 for each bucket in buckets:  
 bucket := logarithm base 10 of (1.0 + bucket / max\_magnitude)  
  
 -- Convert bucket array (raw bar chart data) into a more usable format  
 ranges := new array of FrequencyRange with size equal to size of buckets  
 for i from 0 to size of buckets - 1:  
 lower\_freq := minimum\_frequency\_bark + (i + 0.0) \* bark\_distance  
 upper\_freq := maximum\_frequency\_bark + (i + 1.0) \* bark\_distance  
  
 -- Convert back to Hertz scale  
 lower\_freq := 0.5 + 600.0 \* sinh(lower\_freq / 6.0)  
 upper\_freq := 0.5 + 600.0 \* sinh(upper\_freq / 6.0)  
  
 ranges[i] := FrequencyRange {  
 min\_frequency: integer part of lower\_freq,  
 max\_frequency: integer part of upper\_freq,  
 magnitude: buckets[i]  
 }  
  
 return ranges

function draw\_bar\_chart(bars: array of FrequencyRange):  
 draw\_screen\_background()  
  
 -- Find max magnitude  
 max\_magnitude := 0.0  
 for i from 0 to size of bars - 1:  
 if (bars[i].magnitude > max\_magnitude)  
 max\_magnitude = bars[i].magnitude  
  
 -- Work out scaling  
 bar\_width := screen\_width / (size of bars)  
 bar\_height\_scale := screen\_heigh / max\_magnitude  
  
 for i from 0 to size of bars - 1:  
 bar\_height = bars[i].magnitude \* bar\_height\_scale  
 draw\_rectangle(  
 x: i \* bar\_width,  
 y: 0,  
 width: bar\_width,  
 height: bar\_height  
 )

## Program GUI

The user will interact with the program using a GUI. In order to maximise the potential user-base, I have decided to use a popular C++ GUI library called "wxWidgets", which allows for the creation of GUIs using a singular code-base for Windows, Linux and MacOS, amongst others.

### GUI "screens"

The user will navigate through a variety of "screens" in order to use the program. The GUI flow can be modelled as followed:

* When the program starts, the user must chose to either load an existing playlist (see "Audio Data and Playback"), or create a new one - this is the "choice screen".
* Should the user chose to create a playlist, a new "screen" will be displayed where they can append audio files on the system to a playlist, before saving it.
* Returning back to the initial "choice screen", the user will then chose to load their existing playlist, or optionally create another one for later use.
* After a playlist has been selected, the main "playback screen" will be displayed. Audio playback will begin.
* The user can view the current audio visualisation, as well as audio playback progress.
* A menu will allow the user to configure the current playback and visualisation, such as by adding new audio effects or changing the visualisation settings.
* A separate "screen" can be displayed allowing the user to view current audio effects.
* In this screen, if a user chooses to edit a current audio effect, a new screen will be displayed, showing the various options one can adjust.

#### Summary of "screens"

* "Choice" screen (create new playlist or load existing one)
* "Create playlist" screen
* "Playback" screen
* "Effects list" screen
* "Edit effect" screen

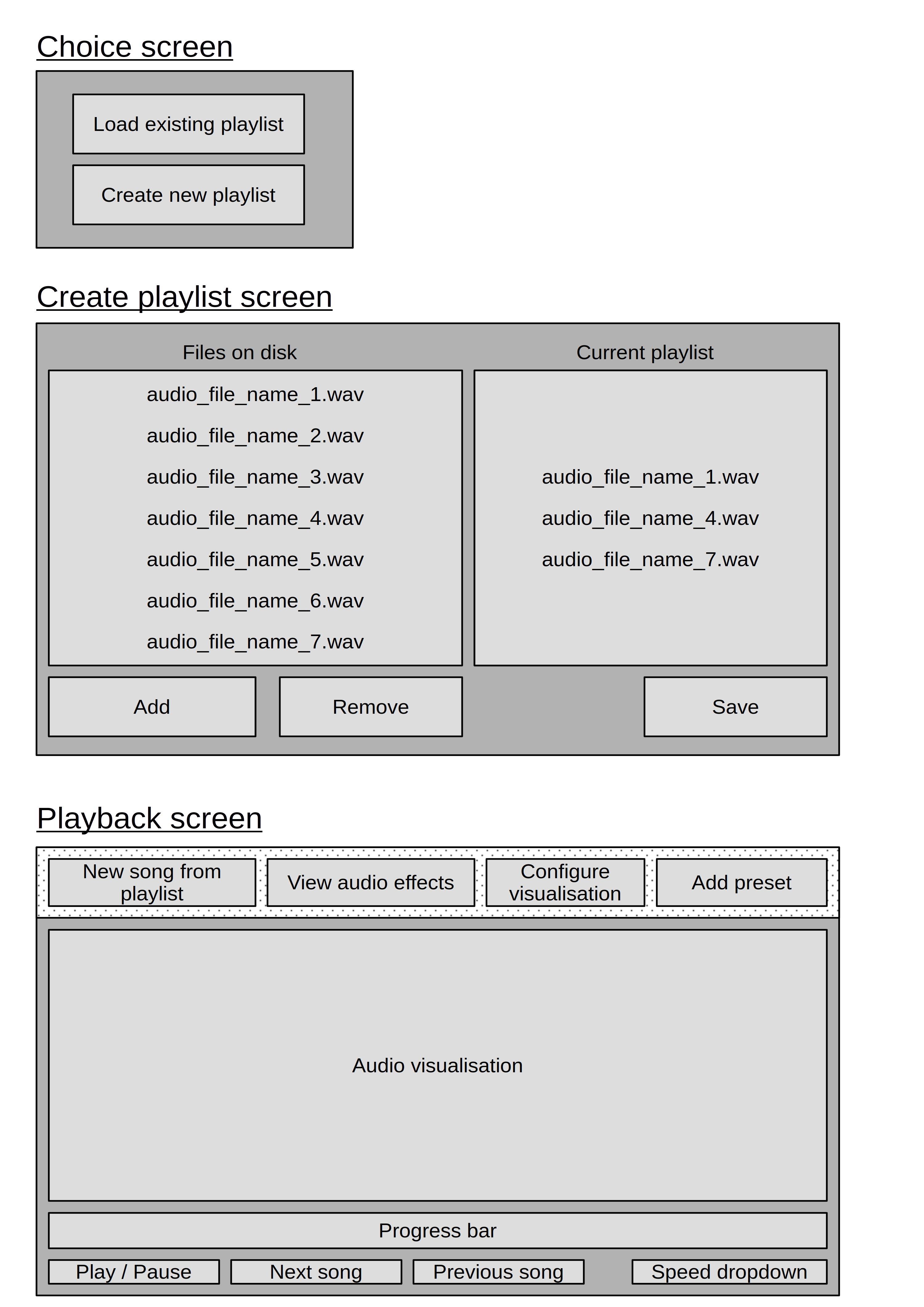
#### Popups

Minor tasks such as selecting a new song from a playlist or modifying visualisation settings will be handled with popup dialogue menus.

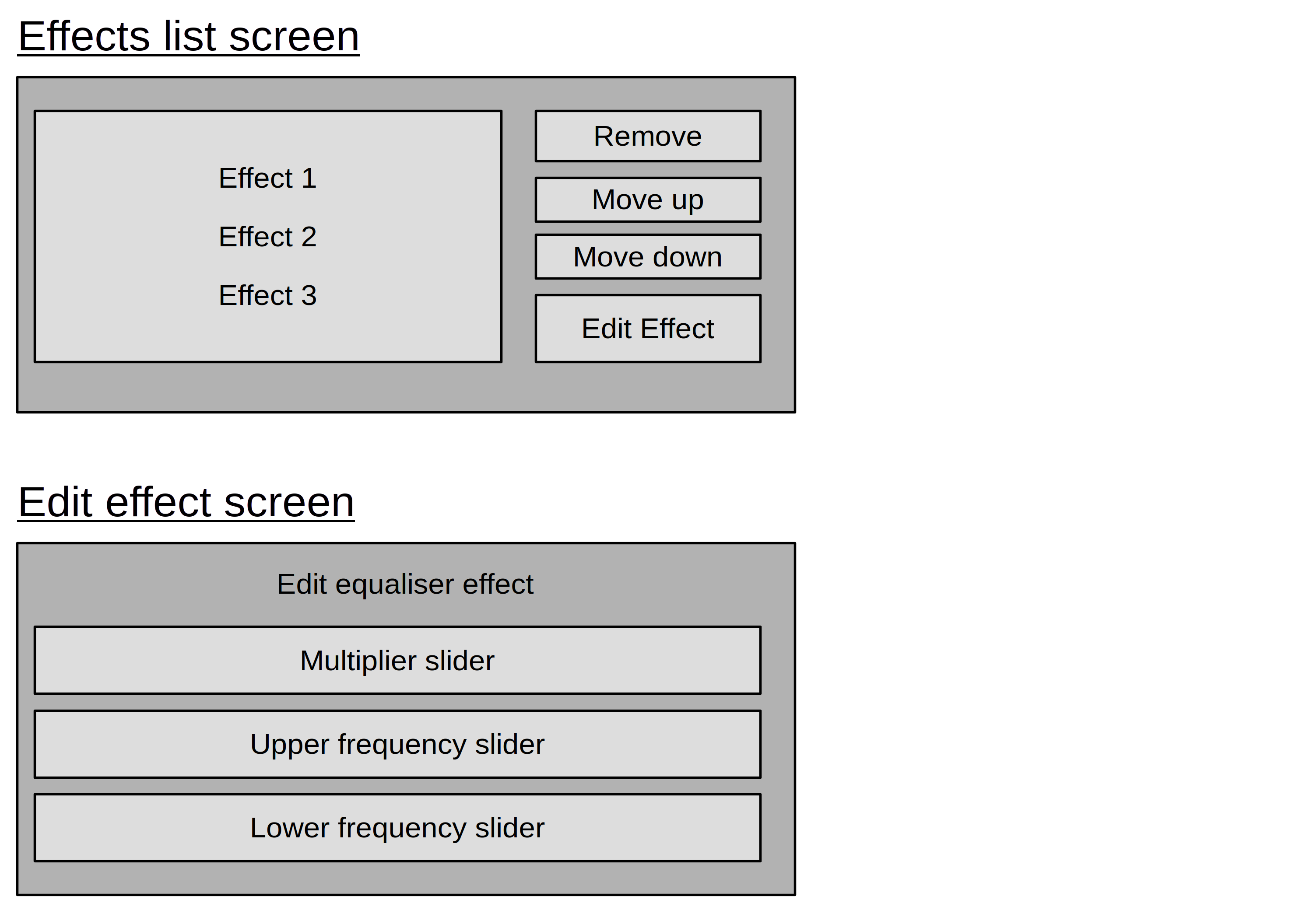


GUI flow for program

### GUI Wireframes



GUI wireframes for main windows



GUI wireframes for ancillary windows

### Relation to the Code

As I am using "wxWidgets" (see above), all GUIs must be programmed directly in code. I have decided, therefore, to create the following classes to abstract away the details of the GUI:

* StartupWindow - implements the initial "choice screen"
* PlaylistWindow - implements the "create playlist screen"
* FileBrowser - responsible for fetching and drawing the list of audio files on the system (used by PlaylistWindow). This is abstracted away as "wxWidgets" does not provide a native "widget" for this, so I must create my own.
* PlayWindow - implements the "playback screen"
* EffectsWindow - implements the "effects list" screen
* PropertiesWindow - implements the "edit effect" screen
* SongSelectionWindow - draws the popup for when the user chooses to select a new song from the current playlist (abstracted as is non-trivial to implement)

## Test Design

In order to meet the objectives described in the analysis section, I have decided to design a variety of measurable tests which will indicate if these objectives have been met. **For convenience, the objectives from above are repeated below:**

### Testing Objective 1

#### Description

"The user must be able to load a collection of audio files known as a "playlist" and then play the audio files contained within in a logical order"

#### Test 1.1

The user must have the option to create a new playlist from a list of audio files on the system. To test this, I will therefore place a variety of audio files on disk and verify they are detected by the program.

#### Test 1.2

I will then test if playlists created in the program can be successfully saved to disk, then loaded back into the program in a sanitised manner. In other words, playlists consisting solely of files which actually exist on disk should be loaded without error, but playlists with invalid audio files should fail to load and notify the user of the error.

#### Test 1.3

After loading a playlist, the audio files contained within must appear "in a logical order", so I will verify that entries in the playlist are organised alphabetically, as is custom in audio programs.

### Testing Objective 2

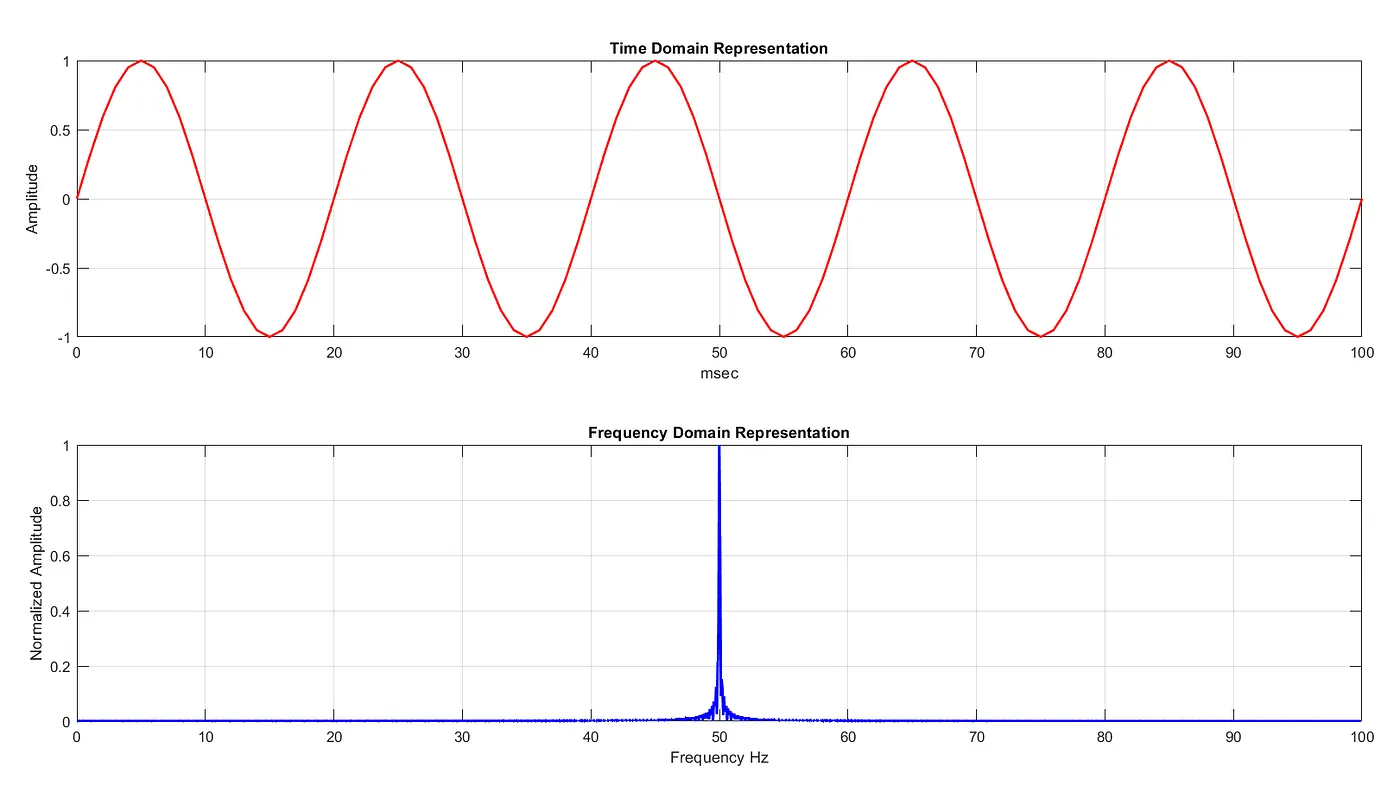
#### Description

"The user must be able to visualise the current audio being played in the frequency domain (i.e. visualise the frequencies)"

As it is difficult for humans to visualise the frequency domain of audio themselves, I will generate audio files consisting of a sine wave of a constant, known frequency. Thus, when the frequency domain is visualised, there should be a single visual peak corresponding to the chosen frequency. By testing the visualisation in this manner across a range of frequencies, it should therefore be possible to deduce if the visualisation is correct.

Obviously the vast majority of audio being played will consist of multiple frequencies. I will therefore combine multiple sine waves in a single audio file (e.g. 500 Hz and 1000 Hz, both playing at the same time). Thus there should be multiple identifiable peaks, each one corresponding to a sine wave frequency chosen.

It should be stressed that human hearing only extends from 20 Hz to 20,000 Hz, and as such any audio visualisation must exclude frequencies outside these ranges.



An example piece of audio, with the time domain shown above and the frequency domain below. If my tests succeed, they should look similar to the first picture (multiple sine waves will of course have multiple peaks of this nature).

2

#### Test 2.1

A correct visualisation of a sine wave is apparent at 500 Hz, with a single peak corresponding to that frequency.

#### Test 2.2

A correct visualisation of a sine wave is apparent at 1,000 Hz, with a single peak corresponding to that frequency.

#### Test 2.3

Sine waves outside the human audible range, at 10 Hz and 100,000 Hz respectively, produce no visible output, as they are inaudible.

#### Test 2.4

A correct visualisation of two sine waves (one at 1,000 Hz and one at 10,000 Hz) is apparent, with two separate peaks corresponding to those frequencies.

#### Test 2.5

A correct visualisation of three sine waves (1,000 Hz, 5,000 Hz and 20,000 Hz ) is apparent, with three separate peaks corresponding to those frequencies.

### Testing Objective 3

#### Description

"The user must be able to modify the audio’s frequency domain (i.e. selectively modify frequencies such as by performing a bass boost)"

#### Test 3.1

The most common use case for this feature will be when adjusting the bass and treble of music. I will therefore play multiple songs in the program with the equaliser effect applied, and verify by ear that the program is able to selectively reduce the range of frequencies chosen.

#### Test 3.2

As a sanity test, I will also ensure that changes made to the audio’s frequency domain are clearly visible in the audio visualisation.[[4]](#footnote-198)

### Testing Objective 4

#### Description

"The user must be able to apply additional "audio effects" to further enhance the music: echo, volume adjustment and noise"

Due to the subjective nature of audio effects, I will ask multiple people from the program’s target audience to provide in-depth feedback on all the audio effects. At least one of these people will be someone experienced in audio processing, so that they are able to assess if the effects provided sound physically accurate (particularly the echo).

#### Test 4.1

A select group from the target audience certify that the echo effect sounds physically plausible

#### Test 4.2

A select group from the target audience certify that the noise effect sounds physically plausible

#### Test 4.3

A select group from the target audience certify that the volume effect adjusts the volume correctly

### Testing Objective 5

#### Description

"The user must be able to configure these "audio effects" individually and also apply pre-made "presets" to quickly reach a desired effect"

#### Test 5.1

All presets present in the application can be loaded successfully and results in the desired effects being applied correctly

#### Test 5.2

All effects can have their various parameters modified, which results in a correct change in the audio playback

### Testing Objective 6

#### Description

"The system must run in real-time on an average school computer"

I will test the performance of the program by running it on a computer in my school, which represents more-or-less average hardware. I will also test it on my school laptop too, which has more modest hardware, to ensure that in all contexts in which a student might run the program, there will be no performance issues. In order to appear to run in real-time, the following must be observed:

1. The audio playback must not slow down [[5]](#footnote-212).
2. The visualisation graphics must update in less than 16.6 ms, such that it is displayed at 60 FPS, which is the most common refresh rate on school hardware. This will be measured by the code itself as it’s running.

#### Test 6.1

The program must run, without any effects applied, in real-time on the specified hardware

#### Test 6.2

The program must run in real-time, for every preset applied separately, in real-time on the specified hardware.

#### Test 6.3

The program must run in real-time, with every effect applied at once, in real-time on the specified hardware.

## Final overview of project hierarchy



Hierarchy chart

# Development and Implementation

Whilst the full source code is available in the appendix, the following section contains the most important parts of the code-base, highlighting the techniques used and technical solutions employed.

## Summary of techniques

For handy reference, below is a table summarising the various techniques used in the codebase. **These are expanded upon in more detail below**.

| Source File | Technique | Purpose |
| --- | --- | --- |
| include/data/atomic\_linked\_list.h | Thread-safe linked list from scratch | Provides data structure for inter-thread communication (see section 2.1) |
| include/data/tree.h | Tree data structure | Used in file browser to represent filesystem structure |
| src/data/merge-sort.cpp | Recursive merge-sort algorithm implementation | To sort playlist files alphabetically |
| src/ui/file\_browser.cpp | Recursively navigate filesystem | Allows the user to browse for audio files on their system |
| src/effects/fft.cpp | Cooley-Tukey Fast Fourier algorithm implementation | To aid in audio procession and visualisation |
| src/ui/audio\_visualiser.cpp | Uses FFT data | Renders audio visualisation |
| include/effects/audio\_effect.cpp | Polymorphic "audio effect" base class (inherited by children) - including use of hashmap | Manages common interface |
| src/effects/echo\_effect.cpp | Inheritance | Echo effect implementation |
| src/effects/equaliser\_effect.cpp | Inheritance and use of Fourier algorithms | Audio equaliser implementation |
| src/effects/noise\_effect.cpp | Inheritance | Noise effect implementation |
| src/effects/volume\_effect.cpp | Inheritance | Volume effect implementation |
| Various files | Use of function callbacks | Used in many places, including preset creation and UI code |
| Various files | Error handling and user input sanitisation | Prevent program from experiencing crashes, security issues and undefined behaviour |

# Testing

# Evaluation

# Appendix

## Copy of Interview Questionnaire

# Code example

void EqualiserEffect::ModifySamples(std::vector<float>& samples, const float frequency) const  
{  
 // Perform FFT to convert to frequency domain  
 FastFourierTransform fft(samples, std::nullopt);  
 std::vector<std::complex<float>>& fft\_output = fft.output;  
  
 ModifyFrequencies(fft\_output, frequency);  
  
 // Perform IFFT to convert back to time domain  
 InverseFourierTransform inverse(fft\_output);  
 std::vector<float> scaled\_real\_components;  
 scaled\_real\_components.reserve(samples.size());  
 for (const auto& c : inverse.output)  
 scaled\_real\_components.emplace\_back(  
 c.real() / (float)inverse.output.size()  
 );  
  
 samples = scaled\_real\_components;  
}

## TODO

* design tests in the design
* narritive: testing: test passed/failed, opinions from others on effects
* evaluation: use testing to say if objectives passed
* evaluation: obejctives pass, but was overall project aim met?
* talk about merge sort
* talk about atomic linked list
* talk about tree
* "fisher yates" shuffle

1. Race conditions occur when one thread tries to read data whilst the other writes to it. If, for example, the GUI thread removed an audio effect from the audio effect list (see above) and freed it from memory whilst the audio thread was applying that same effect, the audio thread would suddenly be reading from invalid memory, likely resulting in a crash or undefined behaviour. [↑](#footnote-ref-91)
2. A mutex is an object that prevents multiple threads from accessing data at the same time. It can be thought of as a lock, which can only be unlocked for one thread at a time. They are preferable to spinlocks as they do not require the CPU to waste cycles waiting for the data to be "unlocked", as instead the thread can suspend itself until the mutex becomes available. [↑](#footnote-ref-92)
3. See above footnote on mutexes [↑](#footnote-ref-95)
4. For example, if one chooses to reduce the bass, then the parts of the visualisation corresponding to the lower frequencies must visually appear smaller in relation to the other frequencies. [↑](#footnote-ref-198)
5. If audio processing takes too long, then the audio callback will be called less frequently than the hardware demands, resulting in a very noticeable slowdown. This must not happen. [↑](#footnote-ref-212)